Configuration and Control of SystemC Models Using TLM Middleware

Christian Schröder
TU Braunschweig, E.I.S.
Germany
schroeder@eis.cs.tu-bs.de

Wolfgang Klingauf
Xilinx Research Labs
San Jose, CA, USA
wolfgang.klingauf@xilinx.com

Robert Günzel
TU Braunschweig, E.I.S.
Germany
guenzel@eis.cs.tu-bs.de

Mark Burton
GreenSocs Ltd.
Cambridge, UK
mark@greensocs.com

Eric Roesler
Intel Corporation
Chandler, Arizona, USA
eric.e.roesler@intel.com

ABSTRACT

With the emergance of ESL design methodologies, frameworks are being developed to enable engineers to easily configure and control models-under-simulation. Each of these frameworks has proven good for its specific use case, but they are incompatible.

ESL engineers must be able to leverage models and tools from different sources in order to be successful. But with today's diversity of configuration mechanisms, engineers spend too much time writing adapters to make tool A cooperate with a model for tool B. We see a need for making the various existing configuration mechanisms cooperate.

We present a solution based on a SystemC middleware. The middleware uses a generic transaction passing mechanism based on TLM-2 concepts and provides inter-operability between the different configuration interfaces in a heterogeneous design. The paper analyses configuration in general and explains the technical consideration for our middleware and shows how it makes the state-of-the-art configuration frameworks inter-operable.

Categories and Subject Descriptors

B.7.2 [**Integrated circuits**]: Design Aids—*Simulation*; I.6.5 [**Simulation and modeling**]: Model Development

General Terms

Design, Standardization

1. INTRODUCTION

The increasing complexity of system-on-chip (SOC) designs raises the requirements on electronic system level (ESL) design methodologies. System level design languages such as SystemC and design methodologies such as transaction-level modeling (TLM) have been introduced, but to reduce complexity of SOC development, reusability becomes important. The problem occurs when trying to connect models from different vendors within one platform, especially if the different models have been constructed using different tool interfaces. The OSCI TLM-2.0 standard overcomes part of this by providing an interoperability interface for describing the memory-mapped communication among models.

However, achieving interoperability at the functional level is just one facet of the problem. The key goal of ESL design is to facilitate architectural exploration using system-level simulators. To this end, ESL engineers need to be able to easily control models in different ways for each simulation run:

- set interconnect properties, such as link widths and frequencies, arbitration schemes, and protocol options *before simulation*
- configure hardware core settings, such as interface widths and functional features *before simulation*
- access device registers, memory contents, and analysis data *during simulation runtime*
- testbench configuration, such as selection of stimuli sequences, enabling/disabling of monitors and scoreboards, etc.

The main contribution of this paper is an open source middleware for SystemC which enables users to configure and control all the (heterogeneous) models in their testbench using one single API or tool, independent of the different configuration and control interfaces or instrumentation frameworks used by the models.

For example, imagine a cell phone design engineer who wants to integrate a CoWare SystemC model of an ARM processor core into the Texas Instruments virtual platform of the OMAP system-on-chip. Connecting the bus interface of the ARM processor model to the communication subsystem of the OMAP model is facilitated with TLM-2. To enable the engineer to configure the various parameters and registers

	SCML [4]	CASI [1]	OVM [3]	System-Python [10]	Intel DRF [13]	Config framew.
Configuration approach	class wrapper	interface	interface	interface	class wrapper	both, internal: class wrapper
Data types	int, unsigned int, double, bool, std::string	std::string	most C++, SystemC number types, strings, objects	strings and number formats	"Instrumented datatype" user implements parame-ter interface	PODs, SystemC types user types (implement interface)
User defined types	-	✓ (user implemented)	✓ (user implemented, help provided)	(unknown)	✓ (instrumented data type)	✓ (template specialized)
File output	XML file	-	-	-	text file	text file
Global registry	✓	✓	(✓) distributed	-	✓	✓
Set default value	✓ (constructor)	user may implement	✓	user may implement	✓	✓ (constructor)
Get value dur-ing runtime by tool	✓	✓	only originally configured value	✓	✓	✓
Runtime value change by owner	✓	✓ (user implemented)	✓	✓ (user implemented)	✓ (user implemented)	✓
Runtime value change by tool	-	✓	-	✓	-	✓
Notify value changes to user	-	-	-	✓ (task call)	✓ (event as member of instr. datatype)	✓ (register callback)
Notify value changes to tool	-	-	-	task-finished event	-	✓ (tool API registers callback)
Param. access with wildcards / regexp.	-	-	(✓)	-	-	✓

Table 1: Review of configuration frameworks.

of the OMAP model with regard to his particular application, Texas Instruments uses a proprietary Python interface called SystemPython. On the other hand, CoWare models use the proprietary SystemC class library SCML to implement configuration parameters and registers.

The example shows that whenever a user wants to mix models from different vendors, he needs to deal with different configuration tools in parallel – an error-prone and inconvenient task. The ARM processor model is controlled with CoWare's Platform Architect tool, but this tool does not know how to deal with the Texas Instruments OMAP models. The OMAP models are controlled via SystemPython scripts, but SystemPython scripts do not enable the user to configure the ARM processor model, since they do not know about SCML. Our example engineer somehow needs to find a way to use both configuration methodologies concertedly.

This paper provides an efficient solution to this problem. Our experiments show that our middleware approach makes possible seamless integration of models written with CoWare SCML, ARM/Carbon CASI, Mentor/Cadence OVM, TI SystemPython, and Intel DRF. We will show that our approach is extremely flexible and can easily be extended to support further configuration and register frameworks.

The main use case discussed in this paper is configuration of heterogeneous (third-party) models *before simulation* and accessing their registers and memories *during simulation*, using a single API or tool. Hence the focus of this work is on model and tool/testbench inter-operation. The experimental results also show that the simulation overhead for accessing device parameters and registers through our middleware is minimal. We therefore believe that the same mechanisms can be efficiently re-used for getting analysis and debug data out of heterogeneous models, thus making the presented approach an ideal candidate to constitute the base for a generic, vendor-independent configuration, control, and analysis standard for SystemC. The latter is the set agenda of OSCI's new CCI working group, and we submitted our work to this group.

2. CONFIGURATION AND CONTROL IN ESL DESIGN

Configuration and control of models in SystemC –which is the ESL language we are focusing on– means to us putting and getting control data to and from a model from outside or inside the design-under-test, before and also during simulation. Configuration is typically understood as setting model properties, e.g. buswidth. In this paper, however, configuration is also used for inspection (get control data/parameters). This is usually achieved by providing a mechanism to read and write model variables from outside the model. Control data exchange between simulator and models may

happen frequently during simulation, e.g. when tracking a model's state; hence the mechanism should be efficient.

Our objective is to link between different configuration and control mechanisms. In this section we will derive requirements for our configuration service to be able to support inter-operation between as many of the existing mechanisms as possible. A review of prominent frameworks such as [1, 3, 4, 10, 12, 13] identified two general approaches to make a model configurable: implementation of special interface methods in the model and class wrappers around model variables.

A model that is equipped with a *configuration interface* [1, 3, 10] has to implement and/or use a special interface containing one or more functions. These functions e.g. announce or declare configurable properties of a module or can get or set the values of those properties, see following example:

```
void set_param(string name, int value);
int  get_param(string name);
```

The second approach for configuring models are *class wrappers* [4, 13], applicable to one or more data types. A model becomes configurable by instantiating an object of this class wrapper which typically replaces the wrapped data type transparently. Within the model the class wrapper acts just like the wrapped object, but also makes the wrapped object accessible for the configuration framework. See an example:

```
class A {
  param<int> my_param;
  param<int> my_reg;
};
```

Both approaches use a pair of an identifier (e.g. a name) and a value to describe a configurable property. We call this pair a *parameter*. Both approaches generally are able to serve a wide list of capabilities: Several data types can be supported, standard C++ and SystemC data types as well as user defined types. Configuration (adding parameters, modifying and reading them) is available before simulation and also can be made available during simulation runtime. External tools as well as simulated models can access the parameters. Default values can be predefined, e.g. in a configuration file or using a database, and configuration snapshots can be exported and stored permanently. A global index of all parameters in a testbench can be created. Notification mechanisms can be implemented to inform registered listeners about changes of parameter values.

There is one major difference between the supported features of the two described implementation approaches: Notifying registered listeners (e.g. the scoreboard in a testbench) about parameter value changes cannot be automated with the interface approach, because with this approach parameters are just plain variables. Whoever changes the value of such a parameter variable – the owner, the test script, or the simulation tool – needs to actively fire the related change event; otherwise the listeners will miss that value change. With the class wrapper approach automation of this feature is no problem because parameter variables are always accessed through their wrappers who then can automatically fire the event.

To be able to support inter-operation of as many configuration mechanisms as possible, our configuration service needs to support both the interface and the class wrapper based approaches. In section 5.3 we will show how the approaches can be mapped to each other.

3. RELATED WORK

The main related work is represented by the commercial ESL tools providing configuration concepts [1, 3, 4, 10, 13]. We have reviewed and compared these frameworks, the overview summary of this work is shown in table 1. The rows list the features that we believe are most important to the user. Note that the rightmost column lists the requirements that we set for our framework.

In addition to the solutions provided by EDA tool vendors, some academic frameworks, libraries or tools for SystemC are available. In the majority of cases the main goal is to analyze a SystemC design on a very high system level. The user may examine modules, ports and signal states of connections between modules, however only few related works focus on pre-simulation and/or runtime model configuration and control.

An interesting approach is gSysC [2]. It provides a graphical interface for simulation control and model analysis. The user can interact with the simulation, show elements that were registered to be shown, but not configure during runtime.

The integrated SystemC debugging environment [5] pursues a related approach: The authors present an integrated environment which provides debugging and visualization however in this paper we concentrate on the configuration aspect.

In [12] Patel et al. propose a sophisticated service oriented architecture for ESL simulation that (amongst others) provides a reflection service enabling runtime analysis of model data. This service provides information about the processes of a module, the ports of modules and their connections and signals. But the authors do not discuss whether non-SystemC data types, like plain class members of an SC_MODULE, are accessible through the reflection service. Also, the authors identify a performance degradation of a factor of about 50. When using the reflection services for gaining analysis data during simulation runtime, such a slowdown is not acceptable.

Another approach is to exploit the classical C++ instruments. In [9], the authors illustrate how the C++ preprocessor, template meta programming, polymorphism, libraries and configuration files can be used to make models configurable. All concepts except for the configuration files require recompilation or relinking. Configuration at simulation runtime is not supported.

IP-XACT [14] contributes to Intellectual Property (IP) reuse by defining a standard exchange format for IP interface descriptions. This covers static configuration of an IP on the source code level: component generators can output

customized IP source code based on the IP-XACT information and user configuration.

This paper deals with dynamic configuration and control during simulation. The presented configuration service, in contrast to the related work, enables users to read and write model parameters and registers before and during simulation, and –as the main contribution– is able to provide access to all models in a testbench using a single API / tool, independent of the individual configuration mechanisms implemented by the models.

4. BASE CONCEPT

To enable a user to access *all* the parameters available in a heterogeneous testbench, regardless of the model-specific mechanisms for accessing them, information about all the available parameters needs to be stored in a central database or service class. This class should provide an interface allowing all kinds of access to the parameters.

A straightforward approach would be to develop a class that provides all configuration services identified during our reviews, providing a rich interface to access all those services, and using adapters to connect to the proprietary model configuration interfaces. This approach is obvious and simple, but there are two major caveats. First, the class would have to manage and control the parameters in parallel or as a replacement for the parameter databases implemented in the original vendor tools, which easily can lead to different sorts of problems. For example, the tool database might not be accessible from outside the tool – then the parameter database needs to be mirrored, which can result in value mismatches. Second, adding support for a new tool that was not covered in our review would require extensions or changes to the rich interface. This might result in modifications required for all adapters.

As an alternative, we propose a bus-like structure that has a single connection to the configuration service class and to which all the models and tools connect, thereby splitting the service provision and connection handling into separate entities. A minimal *middleware* interface is used rather than a rich one. We adopt the OSCI TLM-2 concept of a single transport function that transmits an extensible *generic payload*. In this payload, the service requested from the configuration class is encoded. Thus, the richness of the interface is achieved by an extensible data structure that is passed via the minimal interface.

On top of the middleware, different *user APIs* are provided, giving access to services. The user APIs provide vendor specific function calls, e.g. as required by CoWare SCML models, and translate the function calls into transactions. Internally, *sockets* are used to send the transactions via the middleware (cf. fig. 1). That is, whenever a user wants to integrate a third-party model into an existing simulation platform, an appropriate user API is required that implements the third-party model's configuration interface.

Although we adopt the TLM-2 concept we chose not to use the TLM implementation directly, because our middleware does not model any simulated structure (such as a memory-mapped bus) hence it should not use code designated for

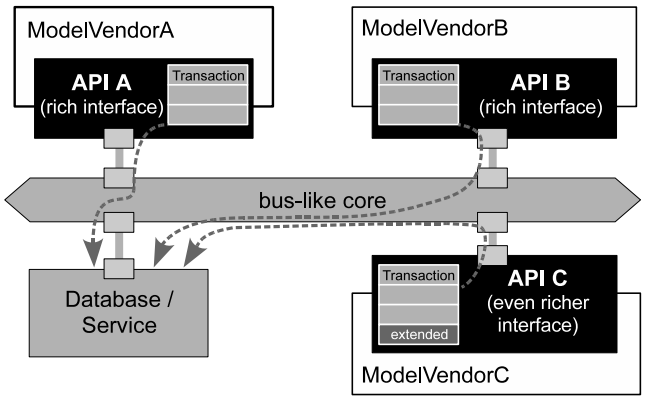

Figure 1: Interface concept divided into rich API adapter (black) and minimal middleware (grey). APIs A and B use the basic transaction, API C uses an extended transaction.

such. In addition, this way we avoid "polluting" any tool (e.g. analysis tools) that automatically shows or interprets TLM-2 sockets and connections. A very valuable TLM-2 feature we can directly adopt is the extension mechanism for the generic payload.

Thanks to the minimal interface approach the middleware becomes service independent as it simply routes transactions from models and tools to the configuration service class. If the implementation of a user API for a new configuration mechanism requires additional information in the transaction, only a new extension for the payload container needs to be implemented, so that existing user APIs do not have to be modified or recompiled.

With the bus-like structure being service independent this approach is not restricted to reading and writing parameter values – it can be used for other services like analysis, debugging or messaging.

4.1 Base concept topology

This section provides details about our implementation of the middleware concepts. The bus-like middleware structure with the minimal interface (the *core* in fig. 2) routes service transactions between one or more initiators and one or more targets. The user APIs connect to the core, and can be used by tools or models to use the services provided by *plugins*.

For example, the configuration plugin provides services to read and write model parameters. Each model in the testbench uses an appropriate user API to announce itself and its parameters to the configuration plugin. A tool uses another user API to search for, get or set model parameters. As a convenience function beyond the capabilities of OSCI TLM-2, the user does not need to bind the internal API's *sockets* manually. They are automatically bound to the middleware during instantiation without user intervention.

Figure 2 exemplifies the use model of our proposal. The functional components communicate over a TLM-2 bus. The components accesses middleware services via configuration user APIs. In this example, the configuration service is pro-

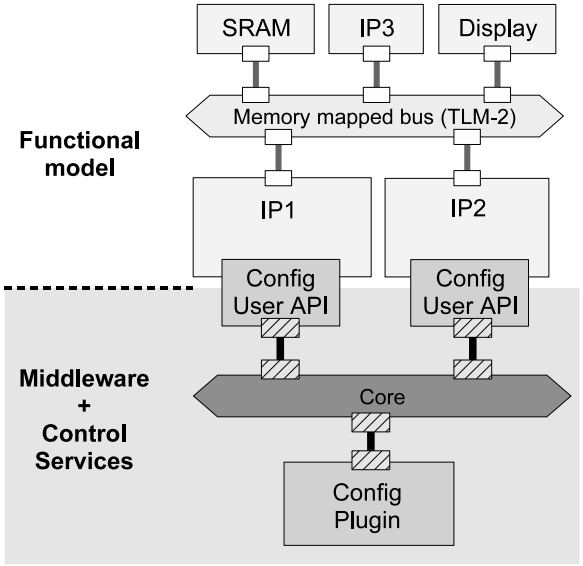

Figure 2: Middleware with service plugin and example user models (IP). Middleware sockets are shaded.

vided by the configuration plugin. The middleware core routes control transactions between APIs and plugins. The only user intervention is to instantiate the appropriate user API in each model (IP1 and IP2 in the example). When the model is using one of the configuration libraries from table 1, this means just replacing one line of code, as will be shown in section 5.2.

4.2 Communication

User APIs and plugins communicate by means of transactions. The payload class ControlTransaction contains attributes for transporting data of variable type, see table 2.

Routing from the API sockets to plugins is based on the unique service ID mService of a plugin, while transactions sent from a plugin to an API are addressed by setting the attribute mTarget which is the pointer address of the API's socket. Plugins should remember the source address transfered as mID attribute to be able to send transactions back to the API if necessary.

The attribute mCmd specifies the command the sender wants execute in the receiver. The meaning of this attribute must be specified by the service plugin and the concerning API (cf. table 3).

The data attributes are designed to carry various kinds of data a service may need to transfer. Nevertheless there may be additional demands which can either be met by using one of the pointer fields to transfer user defined objects or by using payload extensions which are discussed in [11].

The data structure is filled by the sender with data and sent non-blocking to the receiver via a single transport function call (nb_transport). The receiver works on the data structure, changes data if necessary and returns from the function call. Calls to wait are not allowed within the transport calls,

Attribute	Categ.	Description
mService	routing	addresses a service plugin
mTarget	routing	addresses an API
mID	routing	sender's address ID
mError	error	specifies an error
mCmd	cmd	specifies the command
mAnyPointer, mAnyPointer2	data	void pointers for any kind of pointer transmission
mAnyUint	data	any kind of unsigned int
mSpecifier, mValue	data	string specifier/value (e.g. parameter name and value)

Table 2: Control transaction attributes.

because the transactions carry non-functional and therefore timeless information. In the case of an error (e.g. an unknown command was received) the mError attribute of the transaction should be used.

5. CONFIGURATION SERVICE

Using the middleware concepts from the last chapter, we built a flexible configuration service that can deal with models written for a variety of configuration interfaces. Both models that employ the class wrapper approach and models that implement configuration interface methods are supported through different socket implementations.

The following code needs to be added to the testbench in order to make use of the framework. It instantiates the middleware core and the configuration service plugin with its configuration parameter database.

```
#include "control/all.h"
int sc_main(int argc, char *argv[]) {
  ControlCore    core("ControlCore");
  ConfigDatabase cnf_db("ConfigDB");
  ConfigPlugin   cfgPlg("CfgPlg", &cnf_db);
}
```

5.1 Configuration Plugin

The configuration plugin maintains a database of the parameters in the system. In addition to provide read and write access to all parameters in the system, we add the requirement that it shall enable users to register events on parameters. Such events shall be automatically fired whenever the corresponding parameter value has been changed. This will allow to use model variables both as parameters for model configuration and as registers with easily trackable value history. The latter is a very useful feature for debugging and analysis, as it enables users to add introspection support to any model variable. Given this requirement, there are two basic approaches to implement the parameter database:

Store parameter values inside the database. This enables simple support for the class wrapper approach, as class wrappers can then be simple gateways that get/set the value in the database via service transaction whenever the wrapped parameter is read or written to. It also allows simple support for models that use the interface approach, because every call to a configuration interface method can send a service transaction to the plugin. The drawback is that everytime a parameter is read or written a service transaction needs to be executed. This can reduce the simulation

Command	Details
Direction: API → Configuration Service Plugin	
CMD_ADD_PARAM	Adds a parameter *and* sets its default value.
CMD_SET_INIT_VAL	Sets the init value of a parameter.
CMD_GET_PARAM	Gets a pointer to a parameter.
CMD_REMOVE_PARAM	Removes a parameter from the plugin.
CMD_EXISTS_PARAM	Returns if the parameter exists.
CMD_GET_PARAM_LIST	Returns a full parameter list (or a wildcard specified list)
CMD_REGISTER_NEW_ PARAM_OBSERVER	Registers an observer for new added or new implicit parameters.
CMD_UNREGISTER_ PARAM_CALLBACKS	Unregisters all parameter callbacks for the specified observer module.
Direction: Configuration Service Plugin → API	
CMD_NOTIFY_NEW_ PARAM_OBSERVER	Notifies an observer about newly added parameters.

Table 3: Configuration transaction commands

performance. Another drawback is that the ESL tool used for simulation might maintain its own parameter database and if it uses the store-by-value approach as well, it may become difficult to maintain data integrity in the "mirror" database.

Store pointers to class wrappers in the database. This enables simple support of class wrappers. Service transactions will only take place when a tool or model queries the parameter database to retrieve a pointer to a parameter. No transactions are required when a parameter is read or written. The impact on simulation performance is therefore minimal and we chose to use this approach in our implementation. Section 5.3 will show that with the store-by-pointer approach we still can support models that employ the interface method call approach, although this becomes a bit more difficult than with the store-by-value approach.

5.1.1 Implicit parameters

In larger simulations it can be difficult to make sure that all parameters do already exist (i.e., have been instantiated) when a tool is about to configure them, e.g. using a configuration file. Therefore the configuration service should provide the ability to configure not (yet) existing parameters. Therefore the database needs to distinguish between two different parameter modes: when a parameter does not yet exist, setting its (initial) value creates an *implicit parameter* in the database. An implicit parameter has its value stored in the database as a string (because the datatype is not known) and will be converted to an *explicit parameter* when the parameter object is instantiated. Thereby the default value of the parameter will be overwritten with the initial value.

5.2 Basic Configuration API

We created a basic configuration API that provides a native configuration interface on top of our middleware. It is based on class wrappers and can be used to write vendor/tool independent models. In 5.2.1 we show how to instrument model variables with the basic API and 5.2.2 shows how they can be accessed through the middleware.

5.2.1 Model API

To store pointers to parameters in the database a base type needs to be defined that can be stored in a map data structure. The class wrapper gs_param<type> provides all capabilities of the reviewed class wrappers. It is a templated parameter class that works with any kind of data type and derives from a not templated base class gs_param_base that can be stored in the database. The base class provides the access functions to the parameter. The gs_param class is basically a user API and connects its socket automatically to the middleware core, so that simple instantiation like

```
gs_param<int> int_param("int_param", 104);
```

will register the parameter with the database, thereby making it available to tools and other models. The parameter's full hierarchical name is automatically built from the parent SystemC module's name concatenated with the parameter name. The parameter object can be used mostly like the data type it wraps around, since the standard operators such as = and () are overloaded.

The gs_param also allows to register callbacks that are triggered either when the value changes or when the value is read from the class wrapper:

```
GC_REGISTER_PARAM_CALLBACK(
   &int_param, My_Module, callb_function);
```

During instantiation, the class wrapper issues a service transaction to announce the new parameter object to the database. Section 5.3 will show how gs_param can be used to replace other (vendor-specific) class wrappers as well.

5.2.2 Tool API

The configuration user API GCnf_API does the transaction communication with the config plugin using the commands shown in table 3. For example, a call to the API function getPar("MyMod.foo") retrieves the pointer to the parameter named foo from the SC_MODULE MyMod by sending a service transaction with command CMD_GET_PARAM to the plugin.

Listing 1 shows how a tool uses a GCnf_API to register a callback on a parameter.

```
gs_param_base *observedp
   = mGCnfApi->getPar("IP_Native.myp1");
GC_REGISTER_PARAM_CALLBACK(
   observedp, Watch_Tool, callb_function);
```

Listing 1: A tool accessing a parameter (cf. fig. 3)

To be able to use the API in a tool or model, the following global function call needs to be made:

```
cnf_api *mGCnfApi
 = GCnf_Api::getApiInstance(this);
```

Listing 2: Create a user API

5.3 Interfaces to other frameworks

This section discusses how models built for other configuration mechanisms can be connected to our framework.

5.3.1 Class wrapper approach integration

CoWare's SCML property is an example for the class wrapper approach. Properties are configurable objects similar to gs_param from above. To add support for SCML property objects to our configuration service, we simply derive a class wrapper with the correct class name from our own class wrapper. This can be done with a couple of lines of code. We provide stubs for the constructors and functions defined in SCML property. The stubs use the gs_param built-in functions to exercise the desired functionality (read/write parameter value). There are furthermore special functions in SCML property, for example one that returns a CoWare-specific string representation of the wrapped type. These functions need to be implemented as well in the SCML API. Still we consider this an easy task; for the SCML properties it took about an hour to build the replacement.

5.3.2 Interface approach integration

ARM's "Cycle Accurate Simulation Interface" CASI is an example of how interface based configuration can be incorporated into our framework. At start of simulation our CASI API calls the CASI function getParameterList and creates the appropriate number of gs_param<string> instances (CASI parameters are always represented as strings), which make all parameters of the CASI models accessible through our database. Our CASI API registers a pre-read callback and a value change callback with the parameters. When a tool requests the pointer to one of the mirrored CASI parameters and reads its value, the pre-read callback will get triggered. It calls the correct getParameter function in the CASI module, so that the mirror parameter always returns the most recent value of the 'real' parameter which resides in the CASI module. Similarly, when a tool changes the value of a CASI parameter through our configuration service, the value change callback will execute the appropriate setParameter function in the CASI module. Note that when a CASI parameter value gets changed from within its owner model, this will not lead to transactions in the middleware. Our CASI API has less than 100 lines of code.

6. CASE EXAMPLE

To examine if the proposed framework really can bridge the gap between heterogeneous configuration interfaces, we show a case example that combines CoWare models using SCML properties, ARM models using CASI parameters and models using our own gs_param class wrappers. Figure 3 shows the scenario of the case study (the figure shows the configuration topology, not the simulated topology).

We implemented three tool examples and connected them to the middleware: a simple configuration file reader (called Initialize_Tool), a command line parameter query tool, and a watch tool. The tools were used to write initial configurations to all the models in the system, to perform arbitrary read and write accesses to parameters, and to monitor the value change history of the various parameters in the system. In this scenario, all three tools are fully unaware what type (SCML property, CASI parameter, gs_param) of parameter they actually deal with. All three tools were able to successfully access all the parameters of the system. Examples like this can be downloaded at [7].

Listing 3 illustrates the API replacement from the user point of view. It is sufficient to simply use another include file in the model. This is the only change that is required to connect a third party to our configuration service.

```
//#include <scml.h> //CoWare SCML
#include "config/scml.h" //SCML replacement
```

Listing 3: Include either the CoWare SCML or the SCML replacement file

6.1 Performance consideration

Because we assume frequent parameter accesses during simulation runtime (monitoring/analysis/debugging), the framework should be efficient. Neither the existence of configurable parameters nor frequent read or write accesses should slow down the simulation significantly.

We created a performance test (cf. tbl. 4) consisting of two TLM-2 devices: A master exercises 100 million TLM-2 write transactions on a slave which writes the received payload to an unsigned int, an sc_uint or gs_param variables. This means an intensive usage of parameters.

unsigned int	15.48 sec.	1
gs_param<unsigned int>	15.76 sec.	1,02
sc_uint	15.79 sec.	1,02
gs_param<sc_uint>	16,18 sec.	1,05

Table 4: Performance test result.

Table 4 shows that writing values through our middleware to model variables which have been instrumented with our class wrapper (see 5.2.1) is very efficient. The overhead of the middleware is minimal. When integrating models that have been written using vendor-specific approaches, such as CASI or SCML, further effects on simulation performance might occur dependent on the performance of the vendor-specific approaches. For example, CASI parameters are always strings and thus CASI models need to do a string-to-integer conversion whenever an integer parameter is accessed.

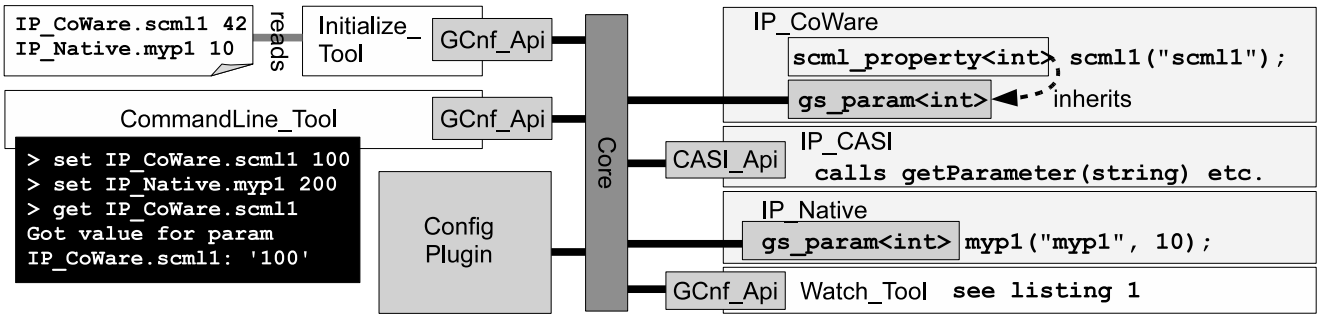

Figure 3: Case study connecting IPs using different configuration mechanisms in one simulation.

7. CONCLUSIONS

In addition to model inter-operability at the functional level which OSCI tackles with the TLM-2 standard, there are more interoperability gaps to bridge. In this paper we presented a configuration middleware that provides interoperability of SystemC model configuration and control interfaces. The middleware automatically creates TLM-2 like connections to pass control messages between models and tools. Different APIs are provided on top of the middleware socket, such as ARM CASI and CoWare SCML. Our case example shows that the effort to install the middleware in existing system models is minimal, and that new user APIs can be added easily to support further vendor-specific IP and tools. A full implementation of the presented concepts and further examples and manuals are available as open-source download [6, 7, 8].

We have shown that our middleware enables users to use one single API and tool to access all parameters in a system, regardless of the (potentially) different instrumentation frameworks that have been used to implement the parameters in the different models in the system. The presented approach considerably simplifies the integration of third-party models into one platform, thus enabling faster and easier architectural exploration.

8. FUTURE WORK

The presented concepts do only work out-of-the-box if one has access to the source code of the third-party models. The include directive for the models' instrumentation class library needs to be re-directed to point to the replacement API in our framework. To overcome this limitation we consider it the tool vendors' responsibility to agree on a generic standard for a model control interface, e.g. in the OSCI CCI working group. We believe the presented work is a significant contribution to the development of such a standard.

The application of the presented concepts is not limited to model configuration. Due to its flexible architecture and in combination with its generic TLM-2 like communication, the middleware offers the possibility to serve as a multi-purpose platform to support further SystemC simulation services such as analysis and debugging. However, further experiments are required to validate the efficiency of this approach.

9. REFERENCES

[1] ARM Ltd. *Cycle-Accurate Simulation Interface (CASI)*, v2.0 edition, 2007.

[2] C. Albrecht, C.J. Eibl, R. Hagenau. A Loosely-Coupled Graphical User Interface for Run-Time Control of SystemC Simulation Models. *International Journal of Simulation Systems, Science & Technology*, May 2006.

[3] Cadence Design Systems, Inc. *OVM Multi-Language Methodology*, v2.0.1 edition, February 2009.

[4] CoWare Inc. *SystemC Modeling Library Manual*, v1.2 edition, May 2007.

[5] F. Rogin, C. Genz, R. Drechsler, S. Rülke. An Integrated SystemC Debugging Environment. *Forum on Specification & Design Languages (FDL)*, Barcelona, Sept. 2007.

[6] GreenSocs. *GreenConfig User's Guide*, v1.2.6 edition, December 2008. http://www.greensocs.com/en/projects/GreenControl/GreenConfig/docs/GCnfUsersGuide.

[7] GreenSocs. GreenConfig framework. 2009. http://www.greensocs.com/projects/GreenControl.

[8] GreenSocs. *GreenControl User's Guide*, v2.0.1 edition, March 2009. http://www.greensocs.com/en/projects/GreenControl/docs/GCUsersGuide.

[9] L. Charest, P. Marquet. Comparisons of different approaches of realizing IP block configuration in SystemC. *proc. IEEE-NEWCAS Conference 2005, pp. 83-86*, June 2005.

[10] M. Berrada, J. Aldis. SystemPython: a Python extension to control SystemC SoC simulations. *GreenSocs meeting presentation, Design and Test in Europe Conference (DATE), Nice*, April 2007.

[11] Open SystemC Initiative (OSCI). *OSCI TLM-2.0 User Manual*, June 2008.

[12] H. D. Patel, D. A. Mathaikutty, D. Berber, and S. K. Shukla. CARH: Service-Oriented Architecture for Validating System-Level Designs. *IEEE Transactions on CAD of Integrated Circuits and Systems, Vol 25, No. 8*, 2006.

[13] J. B. Patrick. Device & Register Framework: Technology & API Overview. *Online presentation material, available at http://www.scribd.com/doc/6316145/Drf-Introduction*, October 2007.

[14] Spirit Consortium. *IP-XACT v1.4: A specification for XML meta-data and tool interfaces*, v1.4 edition, March 2008.